Optimization & Algorithms

MEEC

Project Report

Group: 4

Alexandre Leal (103098) alexandre.b.leal@tecnico.ulisboa.pt
Diogo Ramos (100299) diogo.ramos@tecnico.ulisboa.pt
Diogo Sampaio (103068) diogo.sampaio@tecnico.ulisboa.pt
Francisco Tavares (103402) francisco.carreira.tavares@tecnico.ulisboa.pt

2024/2025 – 1º Semestre, P1

1 Task 1: Theoretical Task

In this task, we demonstrate that the function

$$f_D(w_0, w) = \frac{1}{N} \sum_{n=1}^{N} 1_{R_-}(y_n C_{w_0,w}(x_n))$$

is not convex in the simplified case where $N = 1$ and $D = 1$. For $N = 1$, it simplifies to:

$$f_D(w_0, w) = 1_{R_-}(y_n C_{w_0,w}(x_n)).$$

We define two functions:

$$g(u) = 1_{R_-}(u), \quad f_2(w_0, w) = y_n C_{w_0,w}(x_n).$$

Thus, we express f_D as the composition:

$$f_D(w_0, w) = g \circ f_2.$$

If $g(u)$ is not convex, then f_D is also not convex if f_2 maps to a region where $g(u)$ is not convex. Observing figure 1, we can see that $g(u)$ is not convex.

To demonstrate, using the definition of convexity consider $x = -1$, $y = 1$, and $\alpha = 0.25$:

$$\begin{aligned}
g((1-\alpha)x + \alpha y) \leq (1-\alpha)g(x) + \alpha g(y) \\
\iff \quad & g(0.75 \cdot (-1) + 0.25 \cdot 1) \leq 0.75g(-1) + 0.25g(1) \\
\iff \quad & g(-0.5) \leq 0.75 \cdot 1 + 0.25 \cdot 0 \\
\iff \quad & 1 \leq 0.75.
\end{aligned}$$

This is false; hence, $g(u)$ is not convex, and so f_D is also not convex.

Note: The function f_2 maps $\mathbb{R}$ to $\{+1, -1\}$ since y_n and $C_{w_0,w}$ belong to $\{+1, -1\}$, validating the points used in the example.

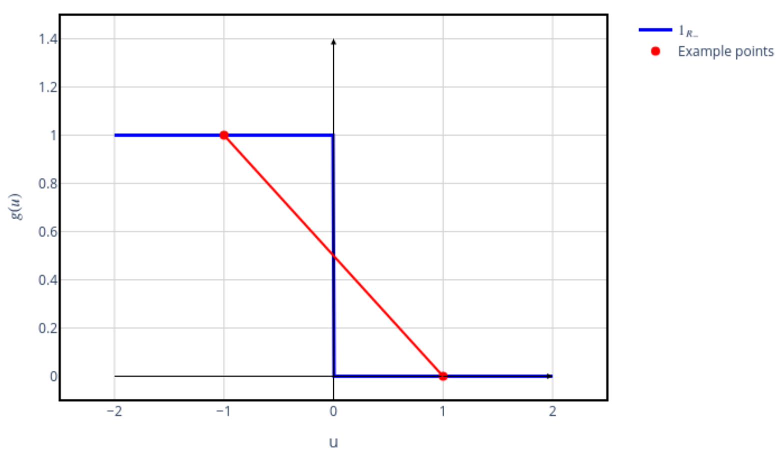

Figure 1: Plot of the function $g(u)$

2 Task 2. [Theoretical Task]

In this task we were asked to show that the function 1_{R_-} is majorized by h, and that $1_{R_-} \leq h(u)$ holds for all $u \in \mathbb{R}$. Furthermore we showed that h is a convex function.

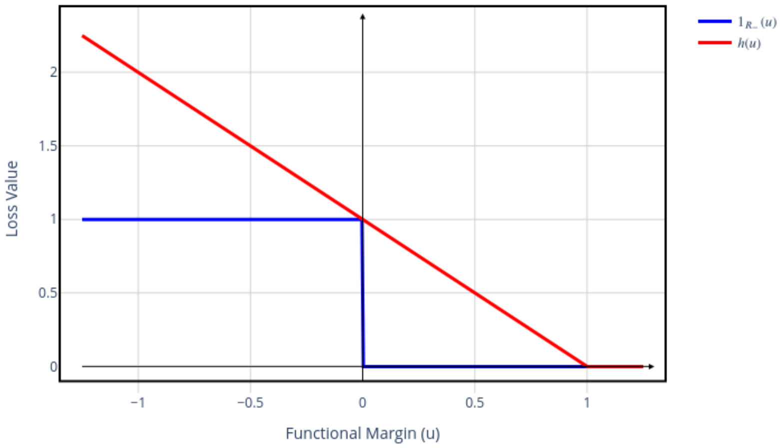

Figure 2: Plot of the indicator function $1_{R_-}(u)$ & hinge loss function $h(u)$

We want to prove that $1_{R_-}(u) \leq h(u)$ for all $u \in \mathbb{R}$. We can describe $1_{R_-}(u)$, and $h(u)$ as:

$$h(u) = \begin{cases} 0, & u \leq 1 \\ 1 - u, & u > 1 \end{cases}$$

$$1_{R_-}(u) = \begin{cases} 1, & u < 0 \\ 0, & u \geq 0 \end{cases}$$

For $u < 0$:

$$\begin{aligned} 1_{R_-}(u) \leq h(u) &\iff 1 \leq 1 - u \\ &\iff 0 \leq -u \iff u \leq 0 \quad True \end{aligned}$$

For $u \in [0; 1[$:

$$\begin{aligned} 1_{R_-}(u) \leq h(u) &\iff 0 \leq 1 - u \\ &\iff u \leq 1 \quad True \end{aligned}$$

For $u \geq 1$:

$$1_{R_-}(u) \leq h(u) \iff 0 \leq 0 \quad True$$

To show that h is a convex function we can observe figure 3, it´s easy to see that no matter which 2 points in h we choose, if we draw a straight line between them, h will be always lower or equal then any point in that line, therefore, h is convex.

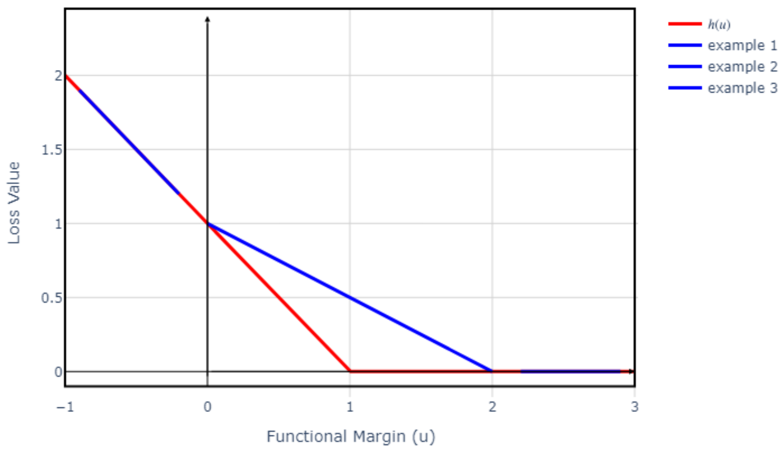

Figure 3: Plot of the hinge loss function $h(u)$ and some examples to show convexity

3 Task 3: Theoretical Task

We are asked to show that the function

$$g_D(\omega_0, \omega) = \frac{1}{N}\sum_{n=1}^{N} h(y_n(\omega_0 + x_n^T\omega))$$

is convex for any N and D.

The function $g_D(\omega_0, \omega)$ can be written as a sum of convex functions in the form:

$$g_D(\omega_0, \omega) = \frac{1}{N}\sum_{n=1}^{N} h(y_n(\omega_0 + x_n^T\omega)) = \frac{1}{N}h(y_1(\omega_0 + x_1^T\omega)) + \cdots + \frac{1}{N}h(y_N(\omega_0 + x_N^T\omega)).$$

If $f_1, \ldots, f_N$ are convex, then $g_D(\omega_0, \omega)$ is convex because a sum of convex functions is convex [slide 12 of module 3 of the theoretical slides]. Since all f_n have the same form, we only need to prove that f_n is convex for all $n \in \{1, \ldots, N\}$.

We define

$$f_n = h(y_n(\omega_0 + x_n^T\omega)),$$

which can be written as $f_n = h(g(\omega_0, \omega))$. To show f_n is convex, we need to check two conditions: 1. $h(u)$ is convex, and 2. $g(\omega_0, \omega)$ is affine [slide 12 of module 3 of the theoretical slides].

First, $g(\omega_0, \omega)$ can be written as

$$g(\omega_0, \omega) = y_n(\omega_0 + x_n^T\omega) = \begin{bmatrix} y_n & y_n x_n^T \end{bmatrix} \begin{bmatrix} \omega_0 \\ \omega \end{bmatrix},$$

where

$$A = \begin{bmatrix} y_n & y_n x_n^T \end{bmatrix} \in \mathbb{R}^{1\times(D+1)}, \quad b = 0, \quad \text{and} \quad x = \begin{bmatrix} \omega_0 \\ \omega \end{bmatrix} \in \mathbb{R}^{D+1}.$$

Thus, $g(\omega_0, \omega)$ is an affine map of the form $g(\omega_0, \omega) = Ax + b$, where A is a matrix and b is a vector.

Since affine functions preserve convexity, and we are given that $h(u)$ is convex (as shown in Task 2), it follows that $f_n = h(g(\omega_0, \omega))$ is convex. Therefore, $g_D(\omega_0, \omega)$ is convex.

4 Task 4. [Theoretical Task]

We can write g as $g_d(w_0, w) + p \cdot (\|w\|_2)^2$, where $p > 0$, which is in the form $t_1 f_1 + t_2 f_2$. Therefore, if g_d is convex (proven in Task 3) and $(\|w\|_2)^2$ is convex, then g is convex.

We can write $(\|w\|_2)^2$ as $f \circ h(w)$, with $h(w) = \|w\|_2$ and $f(u) = (u_+)^2$. We can say this because for $u \geq 0$, $u_+ = u$ and we know that $h(w) \geq 0$ for all w. $f \circ h(w)$ is convex if f is convex and non-decreasing and h is convex. h is an elementary convex function (the l_p norm) [slide 7 of module 3 of the theoretical slides], so it is convex.

$f(u)$ can be described as:

$$\begin{cases} f(u) = u^2, u \geq 0 \\ f(u) = 0, u < 0 \end{cases} \tag{1}$$

Because f is continuous, if both segments are non-decreasing, then f is also non-decreasing. For $u < 0$, f is constant, so it is non-decreasing. For $u \geq 0$, $f'(u) = 2u$. Since $u \geq 0$, it means the slope of f is always positive, which implies that f is non-decreasing. This means $(\|w\|_2)^2$ is convex and therefore g is also convex.

5 Task 5. [Theoretical Task]

To determine if the function g is strongly convex we will evaluate the case where $w = 0$ and $N = 1$. In this range of values we get $g(\omega_0, \omega) = h(y\omega_0)$. If we analyze the range where $yw_0 \geq 1$, g is a null constant, which means it is not strictly nor strongly convex. Equivalently, if we consider $f = g - \frac{-m||\omega_0||^2}{2} = \frac{-m||\omega_0||^2}{2}$, in the same range, for all positive values of m, f is clearly not convex, which equates to g not being strongly convex.

Now using the definition of strongly convex, for g to be SCVX,

$$g((1-\alpha)x + \alpha y) \leq (1-\alpha)g(x) + \alpha g(y)\frac{-n\alpha(1-\alpha)}{2}||x-y||^2 \tag{2}$$

Let's take for example $x = 20$ and $y = 30$ and $\alpha = 0.5$,

$$g(10+15) \leq 0.5g(20) + 0.5g(y)\frac{-n}{8}||x-y||^2 \tag{3}$$

Knowing that for $x > 1$, $g(x) = 0$, $x = 20, y = 30$

$$0 \leq -n\frac{||x-y||^2}{8} \qquad \frac{||x-y||^2}{8} > 0 \tag{4}$$

$0 \leq -n$ is false for all positive values of n therefore g is not strongly convex.

Furthermore, looking at the graph of the hinge loss function at the end of Task 2, we can also see the constant part of the function mentioned above, proving that g is not strongly convex.

6 Task 6. [Numerical Task]

In this task, we solve the following optimization problem numerically, using CVX in Python. The problem involves classifying images of handwritten digits from the MNIST dataset, where the digits 0 and 1 are considered. We use a linear classifier based on the hinge loss function with regularization, as stated in equation (5) of the problem.

The optimization problem we are solving is defined as follows:

$$\min_{w_0,w} \left[\frac{1}{N} \sum_{n=1}^{N} \max(0, 1 - y_n(w^T x_n + w_0)) + \lambda \|w\|_2^2 \right] \tag{5}$$

The first term in the objective function represents the hinge loss, and the second term is an ℓ_2 regularization term to prevent overfitting.

After solving the optimization problem, the following optimal parameters were obtained:

$$w_0 \approx 0.255$$

$$w \approx [0, 0, 0, \ldots, -1.76 \times 10^{-10}, -2.65 \times 10^{-10}, \ldots, 2.43 \times 10^{-3}, \ldots]$$

We also evaluated the classifier error rate f_D on both the training and test datasets using the function:

$$f_D = \frac{1}{N} \sum_{n=1}^{N} \mathbf{1}(y_n \neq \operatorname{sign}(w^T x_n + w_0))$$

The error rates were as follows for λ:

- Training dataset error rate: 0.00%
- Test dataset error rate: 0.12%

In addition to that, we also changed λ to 0.5 to verify our code and noticed, as expected, that f_d evaluates to 0.25% for the training set and 0.25% for the test set.

7 Task 7. [Theoretical Task]

7.1 Task 7 a.

This task is divided into two parts. First, we are asked to show that

$$\tilde{x} = x - P\,\mathrm{sgn}(yw)$$

solves the following equation:

$$\begin{aligned} &\underset{\tilde{x}}{\text{minimize}}\ y(w_0 + \tilde{x}^T w) \\ &\text{subject to } |\tilde{x}_d - x_d| \leq P, \quad \text{for } 1 \leq d \leq D. \end{aligned} \tag{6}$$

To solve this task, we started by expanding the cost function:

$$f(\tilde{x}) = y(\omega_0 + \tilde{x}^T\omega) = y\omega_0 + y\tilde{x}^T\omega,$$

Since $y\omega_0$ is a constant, we wish to minimize $y\sum_{d=1}^{D}\tilde{x}_d\omega$ where $\tilde{x}_d \in [x_d - P; x_d + P]$.

- If $y\omega_d > 0$, we should pick the smallest $\tilde{x}_d$, $\tilde{x}_d = x_d - P$.
- If $y\omega_d < 0$, we should pick the largest $\tilde{x}_d$, $\tilde{x}_d = x_d + P$.
- If $y\omega_d = 0$, any value of ω_d works because $y\tilde{x}_d\omega = 0$

$$\tilde{x}_d = \begin{cases} x_d - P, & \text{if } y\omega_d > 0 \\ x_d + P, & \text{if } y\omega_d < 0 \\ \text{any value}, & \text{if } y\omega_d = 0 \end{cases}$$

From this analysis, we conclude that the optimal solution is given by $\tilde{x} = x - P\,\mathrm{sgn}(yw)$ Next, we need to verify whether this solution satisfies the specified constraints:

$$\begin{aligned} &|\tilde{x}_d - x_d| \leq P, \quad d \in \{1, 2, 3\} \\ &|x - P\,\mathrm{sgn}(yw) - x| = |-P\,\mathrm{sgn}(yw)| = P \leq P \end{aligned} \tag{7}$$

This confirms that $\tilde{x} = x - P\,\mathrm{sgn}(yw)$ solves the optimization problem stated in (1).

7.2 Task 7 b.

In the second part of the task, we are asked to show that for $\tilde{x} = x - P\,\mathrm{sgn}(yw)$, the cost function in (1) evaluates to:

$$y(\omega_0 + x^T\omega) - P||y\omega||_1.$$

Substituting $\tilde{x}$ into the cost function:

$$\begin{aligned}
f(\tilde{x}) &= y\left(w_0 + (x - P\,\mathrm{sgn}(yw))^T w\right) \\
&\iff y\left(w_0 + x^T w - P\,\mathrm{sgn}(yw)^T w\right) \\
&\iff y\left(w_0 + x^T w - P\sum_{d=1}^{D} \mathrm{sgn}(yw_d) w_d\right) \\
&\iff y(w_0 + x^T w) - P\sum_{d=1}^{D} |yw_d| \\
&\iff y(w_0 + x^T w) - P||yw||_1.
\end{aligned}$$

which confirms the evaluation of the cost function.

Thus, we conclude that the derived expression correctly evaluates the cost function for $\tilde{x} = x - P\,\mathrm{sgn}(yw)$.

8 Task 8. [Numerical Task]

Using the result of Task 7, we replaced (x_n, y_n) by $(\tilde{x}_n, y_n)$ with $\tilde{x}_n = x - P\,\text{sgn}(yw)$ and evaluated the function f_d defined in (2), obtaining the classifier of Task 6's error rate on the attacked test dataset. With $\lambda = 0.1$, as used in task 6, and P=0.18, f_d evaluates 43.56% . We also changed λ to 0.5 to verify our code and noticed, as expected, that f_d evaluates 21.88%, which is very close to the expected value 21.9%

9 Task 9. [Numerical Task]

In this section we want to solve the problem of minimizing the effect of an attack on our classifier. This corresponds to solving the problem defined as:

$$\min_{w_0,w} \left[\frac{1}{N} \sum_{n=1}^{N} \max(0, 1 - (y_n(w^T x_n + w_0) - P||y_n w||_1)) + \lambda \|w\|_2^2 \right] \tag{8}$$

with P = 0.18 and $\lambda = 0.1$.

On the training dataset f_D evaluates to 0.75% and in the test dataset f_D evaluates to 0.44%. As for the attacked dataset, f_D evaluates to 2.19%.

We can observe that, while in the test and training sets, the classifier performs slightly worse, which is expected since it is not the ideal classifier for unaltered data, we see a large improvement on the performance on the attacked dataset, since this is the loss function we strive to minimize with this formulation.

10 Task 10. [Numerical Task]

In Task 10, we aim to solve the problem of fitting a piecewise-linear signal to a set of noisy measurements. The signal is modeled as a weighted combination of linear models, and we need to optimize the parameters of these models to minimize the fitting error.

The goal is to minimize the following objective function f:

$$\min_{s_1,r_1,\ldots,s_K,r_K,u_1,v_1,\ldots,u_{K-1},v_{K-1}} \sum_{n=1}^{N} \left(\hat{y}(x_n) - y_n\right)^2 \tag{9}$$

where $\hat{y}(x_n)$ is the predicted value at time x_n, given by:

$$\hat{y}(x_n) = \sum_{k=1}^{K} w_k(x_n)\hat{y}_k(x_n) \tag{10}$$

and y_n represents the measured values.

Each linear model $\hat{y}_k(x)$ is defined as:

$$\hat{y}_k(x) = s_k x + r_k \tag{11}$$

where s_k is the slope and r_k is the intercept for the k-th model.

The weights $w_k(x)$, which determine how the models are combined, are given by the softmax function:

$$w_k(x) = \frac{e^{u_k x + v_k}}{\sum_{j=1}^{K} e^{u_j x + v_j}} \tag{12}$$

These weights $w_k(x)$ ensure that the combination of linear models is smooth across the different regions of the signal.

When solving for the gradient of the objective function, we first get:

$$\nabla f = 2\sum_{n=1}^{N} \left(\hat{y}(x_n) - y_n\right) \cdot \nabla\hat{y}(x_n) \tag{13}$$

where $\nabla\hat{y}(x_n)$ represents the Jacobian matrix J.

The Jacobian matrix J consists of the partial derivatives of $\hat{y}(x_n)$ with respect to the variables s_k, r_k, u_k, and v_k:

$$J_{n,p} = \frac{\partial\hat{y}(x_n)}{\partial p} \tag{14}$$

where p can represent s_k, r_k, u_k, or v_k.
To compute the partial derivatives, we need to evaluate the derivatives of $w_k(x)$ and $w_k(j)$:

1. Derivative of $w_k(x)$ with respect to u_k:

$$
\begin{aligned}
\frac{\partial w_k(x)}{\partial u_k} &= \frac{\partial}{\partial u_k}\left(\frac{e^{u_k x+v_k}}{\sum_{i=1}^{K} e^{u_i x+v_i}}\right) \\
&= \frac{\frac{\partial}{\partial u_k}\left(e^{u_k x+v_k}\right)\cdot\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right) - e^{u_k x+v_k}\cdot\frac{\partial}{\partial u_k}\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right)}{\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right)^2} \\
&= \frac{e^{u_k x+v_k}\cdot x\cdot\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right) - e^{u_k x+v_k}\cdot\left(e^{u_k x+v_k}\cdot x\right)}{\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right)^2} \\
&= \frac{e^{u_k x+v_k}\cdot x\cdot\left(\sum_{i=1}^{K} e^{u_i x+v_i} - e^{u_k x+v_k}\right)}{\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right)^2} \\
&= w_k(x)\cdot x\cdot(1-w_k(x))
\end{aligned}
\tag{15}
$$

2. Derivative of $w_k(x)$ with respect to v_k:

$$
\begin{aligned}
\frac{\partial w_k(x)}{\partial v_k} &= \frac{\partial}{\partial v_k}\left(\frac{e^{u_k x+v_k}}{\sum_{i=1}^{K} e^{u_i x+v_i}}\right) \\
&= \frac{e^{u_k x+v_k}\cdot\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right) - e^{u_k x+v_k}\cdot\frac{\partial}{\partial v_k}\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right)}{\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right)^2} \\
&= \frac{e^{u_k x+v_k}\cdot\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right) - e^{u_k x+v_k}\cdot\left(e^{u_k x+v_k}\right)}{\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right)^2} \\
&= \frac{e^{u_k x+v_k}\cdot\left(\sum_{i=1}^{K} e^{u_i x+v_i} - e^{u_k x+v_k}\right)}{\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right)^2} \\
&= w_k(x)\cdot(1-w_k(x))
\end{aligned}
\tag{16}
$$

3. Derivative of $w_j(x)$ with respect to u_k (for $j \neq k$):

$$
\begin{aligned}
\frac{\partial w_j(x)}{\partial u_k} &= \frac{\partial}{\partial u_k}\left(\frac{e^{u_j x+v_j}}{\sum_{i=1}^{K} e^{u_i x+v_i}}\right) \\
&= -\frac{e^{u_j x+v_j}}{\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right)^2}\cdot e^{u_k x+v_k}\cdot x \\
&= -w_j(x)\cdot w_k(x)\cdot x
\end{aligned}
\tag{17}
$$

4. Derivative of $w_j(x)$ with respect to v_k (for $j \neq k$):

$$\begin{aligned}
\frac{\partial w_j(x)}{\partial v_k} &= \frac{\partial}{\partial v_k}\left(\frac{e^{u_j x+v_j}}{\sum_{i=1}^{K} e^{u_i x+v_i}}\right) \\
&= -\frac{e^{u_j x+v_j}}{\left(\sum_{i=1}^{K} e^{u_i x+v_i}\right)^2} \cdot e^{u_k x+v_k} \\
&= -w_j(x) \cdot w_k(x) \qquad (18)
\end{aligned}$$

Using the results obtained from the derivatives above, we can calculate the following partial derivatives:

1. For u_k:

$$\begin{aligned}
J_{n,k} = \frac{\partial \hat{y}(x_n)}{\partial u_k} &= \frac{\partial}{\partial u_k}\left(\sum_{k=1}^{K} w_k(x_n)\hat{y}_k(x_n)\right) \\
&= \hat{y}_k(x_n)\frac{\partial w_k(x_n)}{\partial u_k} + \sum_{j\neq k} \hat{y}_j(x_n)\frac{\partial w_j(x_n)}{\partial u_k} \\
&= \hat{y}_k(x_n) \cdot w_k(x_n)(1 - w_k(x_n)) \cdot x_n - \sum_{j\neq k} \hat{y}_j(x_n) \cdot w_j(x_n) w_k(x_n) \cdot x_n \\
&= w_k(x_n) \cdot x_n \cdot \left(\hat{y}_k(x_n)(1 - w_k(x_n)) - \sum_{j\neq k} w_j(x_n) \cdot \hat{y}_j(x_n)\right) \qquad (19)
\end{aligned}$$

2. For v_k:

$$\begin{aligned}
J_{n,(K-1)+k} = \frac{\partial \hat{y}(x_n)}{\partial v_k} &= \frac{\partial}{\partial v_k}\left(\sum_{k=1}^{K} w_k(x_n)\hat{y}_k(x_n)\right) \\
&= \hat{y}_k(x_n)\frac{\partial w_k(x_n)}{\partial v_k} + \sum_{j\neq k} \hat{y}_j(x_n)\frac{\partial w_j(x_n)}{\partial v_k} \\
&= \hat{y}_k(x_n) \cdot w_k(x_n)(1 - w_k(x_n)) - \sum_{j\neq k} \hat{y}_j(x_n) \cdot w_j(x_n) w_k(x_n) \\
&= w_k(x_n) \cdot \left(\hat{y}_k(x_n)(1 - w_k(x_n)) - \sum_{j\neq k} w_j(x_n) \cdot \hat{y}_j(x_n)\right) \qquad (20)
\end{aligned}$$

3. For s_k:

$$\begin{aligned}
J_{n,2(K-1)+k} = \frac{\partial \hat{y}(x_n)}{\partial s_k} &= \frac{\partial}{\partial s_k}\left(w_k(x_n) \cdot (s_k x_n + r_k)\right) \\
&= w_k(x_n) \cdot \frac{\partial}{\partial s_k}(s_k x_n + r_k) \\
&= w_k(x_n) \cdot x_n \qquad (21)
\end{aligned}$$

4. For r_k:

$$\begin{aligned} J_{n,(2(K-1)+K)+k} = \frac{\partial \hat{y}(x_n)}{\partial r_k} &= \frac{\partial}{\partial r_k}\left(w_k(x_n) \cdot (s_k x_n + r_k)\right) \\ &= w_k(x_n) \cdot \frac{\partial}{\partial r_k}(s_k x_n + r_k) \\ &= w_k(x_n) \end{aligned} \tag{22}$$

Finally, we can summarize the gradient as:

$$\nabla f = 2 \sum_{n=1}^{N} \left(\hat{y}(x_n) - y_n\right) J \tag{23}$$

10.1 Results for Optimization Variables

After implementing the LM method and running the optimization, the algorithm converged to the following values for the variables:

$$u = \begin{bmatrix} -13.077 \\ -80.105 \\ 88.202 \end{bmatrix} \qquad v = \begin{bmatrix} -2.346 \\ -415.562 \\ -184.845 \end{bmatrix} \qquad s = \begin{bmatrix} -1.202 \\ 3.265 \\ -4.000 \\ 12.126 \end{bmatrix} \qquad r = \begin{bmatrix} -0.077 \\ 23.059 \\ 19.877 \\ -2.653 \end{bmatrix}$$

10.2 Plots

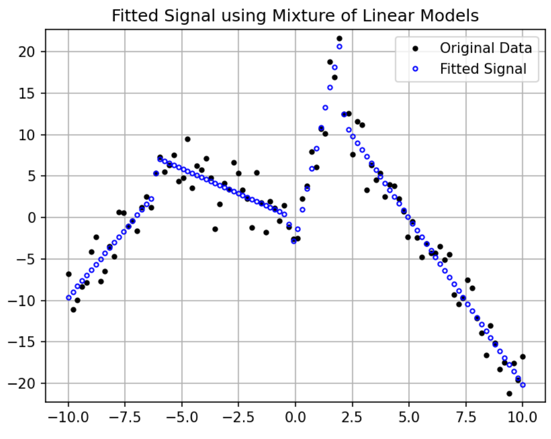

Figure 4: The fitted signal using the mixture of linear models. Each black dot represents a measurement, and the blue circles represent the output of the linear mixture.

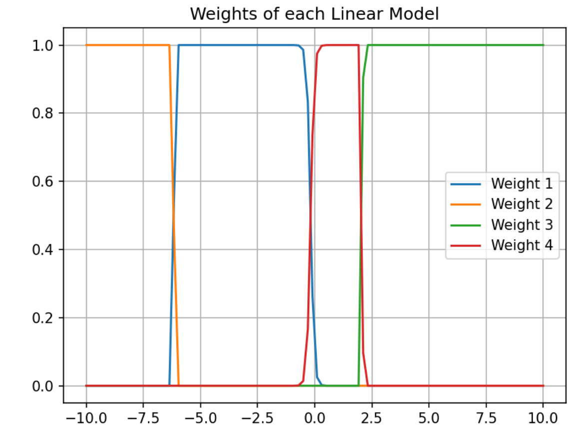

Figure 5: The weights $w_k(x)$ for each linear model. Each color represents a different weight function.

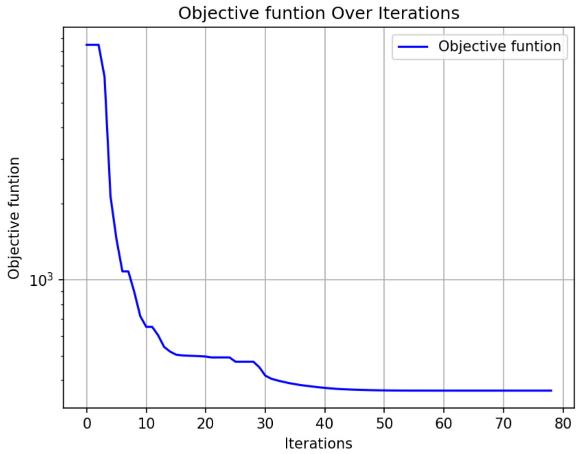

Figure 6: Values of the objective function across the LM iterations.

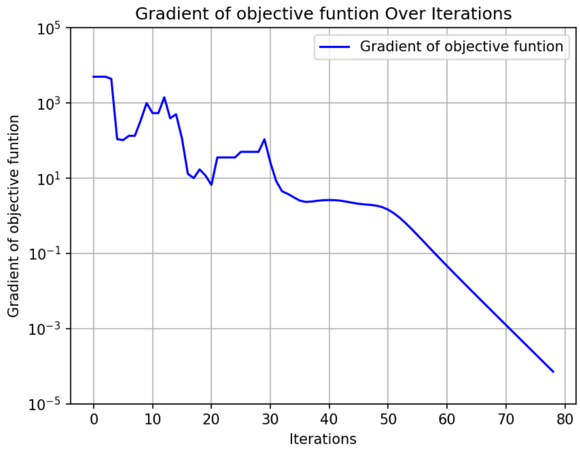

Figure 7: The norm of the gradient of the objective function across the LM iterations.

10.3 Comment on the Results

The Levenberg-Marquardt method converged effectively after 78 iterations, successfully bringing the gradient norm below the set threshold of $\epsilon = 10^{-4}$. As a result, the fitted signal aligns closely with the noisy measurements we started with. The weight functions behaved as anticipated, with each model activating in different regions of the signal, contributing to the overall fit. Throughout the optimization process, the objective function consistently decreased, demonstrating strong convergence behavior, and we observed a clear reduction in the gradient norm over time.

In the end, the final fitted parameters provided an accurate representation of the signal, effectively capturing the distinct linear segments and integrating them into a coherent overall model.

11 Code Developed

11.1 Python Code For Task 6

```python
import cvxpy as cp
import numpy as np
import scipy.io
import matplotlib.pyplot as plt

# Open the data file .mat
data = scipy.io.loadmat('classifier_dataset.mat')

# Extract the variables from the data
traindataset = data['traindataset']
testdataset = data['testdataset']

trainlabels = data['trainlabels'].flatten()  # Flatten to convert into 1D array
testlabels = data['testlabels'].flatten()    # Flatten to convert into 1D array

# Save the number of rows in N (number of samples) and columns in D (number of features)
N, D = traindataset.shape

# Regularization parameter
ro = 0.1

# Define the optimization variables
w0 = cp.Variable()        # Bias term (scalar)
w = cp.Variable(D)        # Weights vector (D-dimensional)

# Define the hinge loss function h(u) = max(0, 1 - u)
hinge_losses = cp.pos(1 - cp.multiply(trainlabels, traindataset @ w + w0))

# Define the objective function: hinge loss + regularization term
objective = cp.Minimize((1/N) * cp.sum(hinge_losses) + ro * cp.norm(w, 2)**2)

# Define the problem and solve it
problem = cp.Problem(objective)
problem.solve()

# Retrieve the optimal parameters
w_optimal = w.value
w0_optimal = w0.value

# Function to evaluate the classifier error rate fD on a given dataset
def evaluate_error_rate(dataset, labels, w0_opt, w_opt):
    predictions = np.sign(dataset @ w_opt + w0_opt)  # Classify
    misclassifications = np.sum(predictions != labels)  # Count errors
    error_rate = misclassifications / len(labels)  # Calculate error rate
    return error_rate

```

```python
# Evaluate the classifier error rate on the training dataset
train_error_rate = evaluate_error_rate(traindataset, trainlabels, w0_optimal
    , w_optimal)
print(f"Training dataset error rate: {train_error_rate * 100:.2f}%")

# Evaluate the classifier error rate on the test dataset
test_error_rate = evaluate_error_rate(testdataset, testlabels, w0_optimal,
    w_optimal)
print(f"Test dataset error rate: {test_error_rate * 100:.2f}%")
```

Listing 1: Python Code to Solve Task 6

11.2 Python Code For Task 8

```python
import cvxpy as cp
import numpy as np
import scipy.io
import matplotlib.pyplot as plt

# Open the data file .mat
data = scipy.io.loadmat('classifier_dataset.mat')

# Extract the variables from the data
traindataset = data['traindataset']
testdataset = data['testdataset']

trainlabels = data['trainlabels'].flatten()  # Flatten to convert into 1D
    array
testlabels = data['testlabels'].flatten()    # Flatten to convert into 1D
    array

# Save the number of rows in N (number of samples) and columns in D (number
    of features)
N, D = traindataset.shape

# Regularization parameter
ro = 0.1

# Define the optimization variables
w0 = cp.Variable()          # Bias term (scalar)
w = cp.Variable(D)          # Weights vector (D-dimensional)

# Define the hinge loss function h(u) = max(0, 1 - u)
hinge_losses = cp.pos(1 - cp.multiply(trainlabels, traindataset @ w + w0))

# Define the objective function: hinge loss + regularization term
objective = cp.Minimize((1/N) * cp.sum(hinge_losses) + ro * cp.norm(w, 2)
    **2)

# Define the problem and solve it
problem = cp.Problem(objective)
problem.solve()

# Retrieve the optimal parameters
w_optimal = w.value
w0_optimal = w0.value

# Function to evaluate the classifier error rate fD on a given dataset
def evaluate_error_rate(dataset, labels, w0_opt, w_opt):
    predictions = np.sign(dataset @ w_opt + w0_opt)  # Classify
    misclassifications = np.sum(predictions != labels)  # Count errors
    error_rate = misclassifications / len(labels)  # Calculate error rate
    return error_rate

P=0.18
```

```python
#####################################################
#make the x~ vector (atacker vector)
#####################################################
N2, D2 = testdataset.shape

x_attack_final=np.empty(shape=(N2, D2))

#For each sample, calculate x~
for k in range(0,N2):
    #calculate y*w
    yw=testlabels[k]*w_optimal
    i=0

    #aply sign function to all elements in the vector yw
    for num in yw:
        if num>=0:
            yw[i]= 1
        else:
            yw[i]= -1
        i+=1

    x=testdataset[k]
    #calculate x~ for this sample
    x_attack= x - P*yw
    #add to matrix with all x~
    x_attack_final[k]=x_attack

#####################################################
#calculate error with attacked input
#####################################################

test_error_rate = evaluate_error_rate(x_attack_final, testlabels, w0_optimal
    , w_optimal)

print(f"Test dataset error rate with attack vector: {test_error_rate *
    100:.2f}%")
```

Listing 2: Python Code to Solve Task 8

11.3 Python Code For Task 9

```python

import cvxpy as cp
import numpy as np
import scipy.io
import matplotlib.pyplot as plt



# Open the data file .mat
data = scipy.io.loadmat('classifier_dataset.mat')

# Extract the variables from the data
traindataset = data['traindataset']
testdataset = data['testdataset']


trainlabels = data['trainlabels'].flatten()  # Flatten to convert into 1D array
testlabels = data['testlabels'].flatten()    # Flatten to convert into 1D array



# Save the number of rows in N (number of samples) and columns in D (number of features)
N, D = traindataset.shape

# Regularization parameter
ro = 0.1

# Define the optimization variables
w0 = cp.Variable()         # Bias term (scalar)
w = cp.Variable(D)         # Weights vector (D-dimensional)

# Define the hinge loss function h(u) = max(0, 1 - u)
hinge_losses = cp.pos(1 - cp.multiply(trainlabels, traindataset @ w + w0))

# Define the objective function: hinge loss + regularization term
objective = cp.Minimize((1/N) * cp.sum(hinge_losses) + ro * cp.norm(w, 2)**2)

# Define the problem and solve it
problem = cp.Problem(objective)
problem.solve()

# Retrieve the optimal parameters
w_optimal = w.value
w0_optimal = w0.value



# Function to evaluate the classifier error rate fD on a given dataset
def evaluate_error_rate(dataset, labels, w0_opt, w_opt):
```

```
    predictions = np.sign(dataset @ w_opt + w0_opt)  # Classify
    misclassifications = np.sum(predictions != labels)  # Count errors
    error_rate = misclassifications / len(labels)  # Calculate error rate
    return error_rate


P=0.18

#####################################################
#make the x~ vector (atacker vector)
#####################################################
N2, D2 = testdataset.shape

x_attack_final=np.empty(shape=(N2, D2))



#For each sample, calculate x~
for k in range(0,N2):
    #calculate y*w

    yw=testlabels[k]*w_optimal
    i=0

    #aply sign function to all elements in the vector yw
    for num in yw:
        if num>=0:
            yw[i]= 1
        else:
            yw[i]= -1
        i+=1

    x=testdataset[k]
    #calculate x~ for this sample
    x_attack= x - P*yw
    #add to matrix with all x~
    x_attack_final[k]=x_attack



w0 = cp.Variable()
w = cp.Variable(D)

ro = 0.1


#Vector manipulation
train_labels_col = cp.reshape(trainlabels, (400,1))
w_T = cp.reshape(w, (1,784))

#Compute yw
product_matrix = train_labels_col @ w_T

#Equivalent to calculating the l1-norm for every row
l1_norm = cp.sum(cp.abs(product_matrix), axis=1)
```

```python

#Define the hinge loss function
hinge_losses_9 = cp.pos(1 - (cp.multiply(trainlabels, traindataset @ w + w0)
    - P * l1_norm))

#Problem definition
objective_9 = cp.Minimize((1/N) * cp.sum(hinge_losses_9) + ro * cp.norm(w,
    2)**2)
problem_9 = cp.Problem(objective_9)
problem_9.solve()

w_optimal = w.value
w0_optimal = w0.value


#Result Evaluation
train_error_rate = evaluate_error_rate(traindataset, trainlabels, w0_optimal
    , w_optimal)
print(f"Training dataset error rate: {train_error_rate * 100:.2f}%")

test_error_rate = evaluate_error_rate(testdataset, testlabels, w0_optimal,
    w_optimal)
print(f"Test dataset error rate: {test_error_rate * 100:.2f}%")

attack_test_error_rate = evaluate_error_rate(x_attack_final, testlabels,
    w0_optimal, w_optimal)
print(f"Test dataset error rate with attack vector: {attack_test_error_rate
    * 100:.2f}%")
```

Listing 3: Python Code to Solve Task 9

11.4 Python Code For Task 10

```python
import numpy as np
import scipy.io as sio
import matplotlib.pyplot as plt


# Load the .mat file
def load_data(file_path):
    data = sio.loadmat(file_path)
    X = data['x']  # Input data (X values)
    Y = data['y']  # Target data (Y values)
    U = data['u']  # ...
    V = data['v']  # ...
    S = data['s']  # ...
    R = data['r']  # ...
    return X.flatten(), Y.flatten(), U.flatten(), V.flatten(), S.flatten(),
   R.flatten()



# Mixture of Linear Models
def mixture_model(x, u, v, s, r, K, N):
    y_pred = np.zeros(N)
    W = np.zeros((N, K))

    alphas = np.zeros((N, K))
    for k in range(K-1):
        alphas[:, k] = u[k] * x + v[k]

    # Centering to prevent overflow
    max_alpha = np.max(alphas,axis=1,keepdims=True)
    exp_alphas = np.exp(alphas - max_alpha)

    # Normalizing weights
    for k in range(K):
        W[:, k] = exp_alphas[:, k] / np.sum(exp_alphas, axis=1)

    # Combine predictions from all K models
    for k in range(K):
        y_pred += W[:, k] * (s[k] * x + r[k])

    return y_pred, W



# Compute partial derivatives (Jacobian elements)
def compute_jacobian_partial(x, u, v, s, r, W, k, K, N, param):
    if param == 's':
        return W[:, k] * x
    elif param == 'r':
        return W[:, k]
    elif param == 'u':
        J_partial_u = np.zeros(N)
        for i in range(N):
```

```python

            k_term = W[i, k] * (1 - W[i, k]) * x[i]*(s[k] * x[i] + r[k])# derivative of Wk*Yk

            j_term = W[i, k]*x[i]*sum(W[i, j] * (s[j] * x[i] + r[j]) for j in range(K) if j != k) # sum of derivative of Wj*Yj

            J_partial_u[i] = k_term - j_term


        return J_partial_u

    elif param == 'v':
        J_partial_v = np.zeros(N)
        for i in range(N):

            W_term = W[i, k] * (1 - W[i, k])*(s[k] * x[i] + r[k])# derivative of Wj*Yj

            sum_term = W[i, k]*sum(W[i, j] * (s[j] * x[i] + r[j]) for j in range(K) if j != k) #sum of derivative Wj*Yj

            J_partial_v[i] = W_term - sum_term

        return J_partial_v
    else:
        raise ValueError(f"Invalid parameter: {param}")

# Compute the gradient and Jacobian
def compute_Jacobian(x, u, v, s, r, W, K, N):
    J = np.zeros((N, 4 * K - 2))  # Jacobian

    # Compute partial derivatives for uk and vk
    for k in range(K - 1):
        J[:, k] = compute_jacobian_partial(x, u, v, s, r, W, k, K, N, 'u') # Jacobian for u_k
        J[:, (K - 1) + k] = compute_jacobian_partial(x, u, v, s, r, W, k, K, N, 'v')  # Jacobian for v_k

    # Compute partial derivatives for sk and rk
    for k in range(K):
        J[:, 2*(K - 1)+ k] = compute_jacobian_partial(x, u, v, s, r, W, k, K, N, 's')  # Jacobian for s_k
        J[:, 2*(K - 1)+ K + k] = compute_jacobian_partial(x, u, v, s, r, W, k, K, N, 'r')  # Jacobian for r_k

    return J


# Levenberg-Marquardt optimization
def levenberg_marquardt(x, y, u, v, s, r, K, N, max_iter=5000, epsilon=1e-4, lambda_init=1.0):
    lambda_ = lambda_init  # Damping parameter
    residuals_list = []  # To store sum of squared residuals for ploting
    grad_obj_func_list = [] # To store gradient of objective function for
```

```
ploting

    for iter in range(max_iter):

        # Compute model prediction and weights
        y_pred, W = mixture_model(x, u, v, s, r, K, N)
        # Combine the u, v, s, r vectors into a single column vector
        param_vector = np.concatenate([u, v, s, r]).reshape(-1, 1)

        # Compute the residual and objective function
        residual = y_pred - y
        obj_func = np.sum(residual ** 2)
        residuals_list.append(obj_func)
        # Compute the full Jacobian matrix (gradients)
        J = compute_Jacobian(x, u, v, s, r, W, K, N)

        # Construct the A matrix (Jacobian and regularization term)
        sqrt_lambda = np.sqrt(lambda_)

        # Create a (4*K-2)x(4*K-2) identity matrix
        identity_matrix = np.identity(4*K-2)

        # Multiply the identity matrix by sqrt_lambda to get the final
sqrt_lambda matrix
        sqrt_lambda_matrix = sqrt_lambda * identity_matrix

        # Combine the jacobian with the square root of the scalar
        A = np.vstack([J, sqrt_lambda_matrix])


        J_param = J @ param_vector  # Multiply jacobian by parameters vector
        b_top = J_param - residual.reshape(-1, 1)  # Subtract residual from
each row

        # Bottom part: sqrt_lambda * param_vector
        b_bottom = sqrt_lambda * identity_matrix @ param_vector  # Shape
will be 14 x 1

        # Combine the top and bottom parts
        b = np.vstack([b_top, b_bottom])

        # Solve the least-squares problem for min
        minimized_parameters,_ , _, _ = np.linalg.lstsq(A, b, rcond=None)

        # Compute the candidate parameters with min
        u_new, v_new, s_new, r_new = update_parameters(u, v, s, r,
minimized_parameters, K)
        # Compute new prediction and objective function with updated
parameters
        y_pred_new, W_new = mixture_model(x, u_new, v_new, s_new, r_new, K,
N)
        obj_func_new = np.sum((y_pred_new - y) ** 2)


        gradient = 2 * J.T @ residual
```

```
        # Check stopping criterion
        grad_obj_func_list.append(np.linalg.norm(gradient))
        if np.abs(np.linalg.norm(gradient))< epsilon:
            print(f"Converged at iteration {iter}")
            break
        # Check if the step is valid

        if obj_func_new < obj_func:  # Valid step
            u, v, s, r = u_new, v_new, s_new, r_new
            lambda_ *= 0.7  # Decrease lambda


        else:  # Null step
            lambda_ *= 2.0  # Increase lambda


    # Plot the results

    plot_results(x, y, u, v, s, r, y_pred, W,residuals_list,
   grad_obj_func_list)
    return u, v, s, r

# Update the parameters (u, v, s, r) with the delta step from LM
def update_parameters(u, v, s, r, minimized_parameters, K):

    # Ensure the deltas are treated as 1D vectors instead of 2D
    u_new = minimized_parameters[:K - 1].flatten()  # Flatten in case it's
   higher dimensional
    v_new = minimized_parameters[K - 1:2 * K - 2].flatten()
    s_new  = minimized_parameters[2 * K - 2:3 * K - 2].flatten()
    r_new  = minimized_parameters[3 * K - 2:].flatten()

    # Update the parameters with the corresponding deltas


    return u_new , v_new , s_new , r_new

# Plot the fitted signal and the weights
def plot_results(x, y, u, v, s, r, y_pred, W,residuals_list,
   grad_obj_func_list):

    plt.figure()
    plt.plot(x, y, 'ko', markersize=3, label='Original Data')
    plt.plot(x, y_pred, 'o', markerfacecolor='none', markeredgecolor='blue',
     markersize=3, label='Fitted Signal')
    plt.ylim(min(y) - 1, max(y) + 1)
    plt.xlim(min(x) - 1, max(x) + 1)
    plt.title('Fitted Signal using Mixture of Linear Models')
    plt.legend()
    plt.grid(True)
    plt.show()

    plt.figure()
    for k in range(W.shape[1]):
        plt.plot(x, W[:, k], label=f'Weight {k+1}')
```

```
    plt.title('Weights of each Linear Model')
    plt.legend()
    plt.grid(True)
    plt.show()

    plt.figure()
    plt.plot(range(len(residuals_list)), residuals_list, 'b-', label='
    Objective funtion')
    plt.xlabel('Iterations')
    plt.ylabel('Objective funtion')
    plt.title('Objective funtion Over Iterations')
    plt.legend()
    plt.grid(True)
    plt.yscale('log')
    plt.show()

    plt.figure()
    plt.plot(range(len(grad_obj_func_list)), grad_obj_func_list, 'b-', label
    ='Gradient of objective funtion')
    plt.xlabel('Iterations')
    plt.ylabel('Gradient of objective funtion')
    plt.title('Gradient of objective funtion Over Iterations')
    plt.legend()
    plt.ylim(0.00001, 100000)
    plt.grid(True)
    plt.yscale('log')
    plt.show()




# Example usage
file_path = 'lm_dataset_task.mat'  # Replace with actual path
# Initial values for u, v, s, r
x, y, u, v, s, r = load_data(file_path)


K = len(s)  # Number of models
N = len(x)  # Number of data points

# Run the LM optimization
u_opt, v_opt, s_opt, r_opt = levenberg_marquardt(x, y, u, v, s, r, K, N)
#print results
print(u_opt)
print(v_opt)
print(s_opt)
print(r_opt)
```

Listing 4: Python Code to Solve Task 10